Published in final edited form as:
Concurr Comput. 2014 March 25; 26(4): 854–865. doi:10.1002/cpe.2928.

Parallel classification and feature selection in microarray data using SPRINT

Lawrence Mitchell[1,*,†], **Terence M. Sloan**[1], **Muriel Mewissen**[2], **Peter Ghazal**[2], **Thorsten Forster**[2], **Michal Piotrowski**[1], and **Arthur Trew**[1]

[1]EPCC, School of Physics and Astronomy, University of Edinburgh, Edinburgh, EH9 3JZ, UK

[2]Division of Pathway Medicine, University of Edinburgh, Medical School, 49 Little France Crescent, Edinburgh, EH16 4SB, UK

SUMMARY

The statistical language R is favoured by many biostatisticians for processing microarray data. In recent times, the quantity of data that can be obtained in experiments has risen significantly, making previously fast analyses time consuming or even not possible at all with the existing software infrastructure. High performance computing (HPC) systems offer a solution to these problems but at the expense of increased complexity for the end user. The Simple Parallel R Interface is a library for R that aims to reduce the complexity of using HPC systems by providing biostatisticians with drop-in parallelised replacements of existing R functions. In this paper we describe parallel implementations of two popular techniques: exploratory clustering analyses using the random forest classifier and feature selection through identification of differentially expressed genes using the rank product method.

Keywords

HPC; Genomics; Parallel programming

1. INTRODUCTION

The introduction of microarray-based technology to biology has established a new generation of highly parallel and high throughput experiment platforms [1]. These can measure tens of thousands of gene transcripts or millions of other genomic sequences in a given biological sample in a very short time. This allows researchers to study which parts of a human (or other) genome are active in, or affected by, a given treatment or phenotype in a population. These data sets are characterised by large size, unbalanced dimensionality and analytical complexity [2]. Following image processing and initial data processing, an example small analysis-ready transcriptomic data set may consist of an $N_g \times N_s$ data matrix of size 25 000 (genes) by 20 (samples). A large genotyping data set may consist of 2 million single-nucleotide polymorphism probes (SNPs) by 2000 samples. Dimensionality is of

[*]Correspondence to: Lawrence Mitchell, EPCC, School of Physics and Astronomy, University of Edinburgh, Edinburgh, EH9 3JZ, UK . [†]lawrence.mitchell@ed.ac.uk.

Europe PMC Funders Author Manuscripts

concern to most analysis approaches, with the number of variables (genes, SNPs, sequences) vastly outweighing the number of observations [3].

In addition to analyses focusing on the behaviour of individual genes or genomic sequences, exploratory analyses are typically rooted in machine learning algorithms. These can be subdivided into unsupervised and supervised clustering. Unsupervised clustering aims to cluster groups of genes or samples that exhibit strong similarities, as this may indicate genomic co-regulation or patient phenotype sets. Supervised clustering aims to classify the genomic features of a previously classified data set to predict the class (diseased or healthy, at risk or not, affected or unaffected) of a previously unknown patient sample.

If we perform these analyses on the complete data, it may be difficult to identify patterns in the data due to the (potentially millions) of variables in each observation. Before clustering analysis, we may therefore decide to restrict ourselves to clustering on a subset of the measured variables. But which ones? Feature selection methods allow us to identify an important subset of variables from our microarray data. For example, in microarray transcription studies, we are looking for strongly differentially expressed genes. A popular, biologically motivated metric for feature selection is the rank product method [4], of which we describe a parallelisation in this paper.

As the data sets get larger, analyses of this post genomic data are increasingly running up against limits in serial computational power [5, 6]. Although existing computational techniques are no less valid than for smaller data sets, the time to perform analyses has risen significantly. Exploiting parallel architectures to speed up analyses is possible but often requires domain specific knowledge and programming experience from the user.

The SPRINT [7] project aims to implement a library that biostatisticians can use to exploit HPC systems while requiring only minimal changes to their existing analysis workflows. SPRINT provides drop-in replacements for a number of computationally expensive R functions that were identified as important in a user requirements survey of the bioinformatics community [8]. These drop-in replacements are parallelised using the Message Passing Interface (MPI) [9], with data distribution carried out transparently from the end-user's point of view. For a more detailed description of the SPRINT architecture, see [10]. Users of SPRINT write R analysis scripts as they would have performed previously for serial analysis. To compute in parallel, the R script must be executed like any other parallel MPI task (e.g. `mpiexec -n 16 R -f script.R`)

In this paper, we describe the parallelisation of two analysis tools identified through this survey: the random forest classifier [11] for clustering analysis and the rank product method [4] for feature selection. An R implementation of the former is provided in the `randomForest` package [12], the latter is available in the `RankProd` package [13].

1.1. Exploiting parallelism in R

R [14, 15] itself has no built in parallel features. However, a number of open source packages add parallel features at different abstraction levels, all are available through the Comprehensive R Archive Network[‡] (CRAN). Rmpi [16] provides wrappers around the

MPI message passing libraries [9]. This approach offers the most flexibility but requires that programmers have good knowledge of parallel programming: existing analysis scripts require significant modifications to take advantage of parallel features. In addition to these extensions, there are also a number of R packages that allow for parallelisation of problems that have no data dependence. For example using bootstrap resampling to calculate statistical properties of a data set. In these latter cases, all calculations are independent so the work can be carried out in parallel and results gathered together at the end of the computation. R packages that provide some of this functionality include but are not limited to R/Parallel [17], `taskPR` [18] and `foreach` [19]: the CRAN task view in high performance computing[§] lists many others.

These packages provide a working and often efficient entry route into parallelising R code for computationally hard analysis problems, but they require a level of programming skill that may not be matched by growing numbers of statisticians, bioinformaticians and biologists who are now being presented with easily generated large data sets.

In contrast to the packages providing generic parallel features, the SPRINT framework does not expose any low-level parallel functionality to the user; instead, it is an R library providing an interface to parallelised statistical algorithms. Adding a new function involves writing the parallel algorithm (typically in C or Fortran) and an interface function in R [7]. This approach makes for much better ease of use by bioinformaticians: in most cases, their existing analysis scripts will need only minor changes. A further benefit of this approach is that individuals can access a (growing) repository of parallelised functions, rather than recreating the method from scratch each time. Although only a small number of functions have been parallelised, in our experience, there are only few stages of typical analysis workflow that require parallel speedup.

2. THE RANDOM FOREST CLASSIFICATION METHOD

The random forest algorithm is an ensemble tree classifier that constructs a forest of classification trees from bootstrap samples of a data set. The classification of an unseen datum is the modal class of the datum over all trees in the forest. The random forest classifier is a popular one for microarray data because it does not just give the classification of a datum but can also indicate to the user which genes are important for the classification along with an estimate of the classification error [11].

2.1. Building a single tree

The construction of a decision tree is conceptually straightforward. The tree consists of *nodes* each of which splits the data set based on the value of some attribute. From the root node (which contains the full data set), we recursively split into children until some terminating criterion is reached. This might be that the number of cases at a particular node drops below a threshold value or that all cases share the same class. Such a terminal node is

[‡] http://cran.r-project.org/
[§] http://cran.r-project.org/web/views/HighPerformanceComputing.html

called a *leaf node*. Once the tree is constructed, we can classify an unseen case by sending it down the tree from the root node. The predicted class is the leaf node it ends up in.

The splitting criterion used in the random forest method is the same as that of the Classification And Regression Tree [20] algorithm: we minimise the *Gini index* of the split [20]. This is a measure of how well the split does in reducing the mixing of classes in the child nodes: a good split will do a better job of reducing the mixing than a bad split. An alternative to the Gini index is to pick the split which minimises the information entropy [21,22].

Most decision tree classifiers are deterministic. That is, for a particular data set, they will always produce the same tree. The trees grown by the random forest algorithm do not have this characteristic. This is because of the way variables are picked to split on at each node. Rather than choosing the best split amongst all variables at each node, a fixed size random subset of all the variables is chosen at each node. Hence, an identical datum may be classified differently by two trees grown on the same data set.

2.2. Growing the forest

To construct a random forest, we generate some large number (typically thousands) of bootstrap samples of our original data set and grow a classification tree for each of these samples using the method described earlier. The data in the original data set are then classified by sending them down each of these trees and selecting the modal class. That is, if 100 trees vote that case A is infected, whereas 900 trees vote that it is a control, case A is classified as a control.

3. THE PARALLEL IMPLEMENTATION

We have two options for parallelising generation of random forests, either parallelising the bootstrap phase or the generation of a single tree. The former is a task parallel approach, distributing the bootstrap samples amongst parallel processes and combining results at the end. The latter option of parallelising single tree growth is more complicated, a number of existing algorithms for growing decision trees in parallel do exist [23-25]. All these approaches are designed for the data one encounters in the social sciences. That is, very many cases (hundreds of thousands or millions) but only a small number of variables (tens or hundreds) describing each case. The algorithms exploit the parallelism available in the cases, dividing them between parallel processes.

Unfortunately, these algorithms do not map well onto microarray data, because the number of cases is low (typically tens or hundreds) whereas the number of variables is large (typically thousands or millions). Furthermore, because each split in a tree only considers a subset of all the variables, if we were to parallelise across variables (rather than cases), the load balance would be poor.

As a result of these issues, and because the typical microarray data set is easily small enough to fit in the memory of a single R process, we decided not to pursue a data parallel approach. Instead, we have implemented a task parallel random forest generator for SPRINT.

Having decided on a task parallel implementation, and because one of the aims of the SPRINT project is to provide a drop in parallel replacement for a serial code, we decided to reuse the existing R code for serial random forest generation. Our implementation therefore exactly mimics the calling conventions of the serial code. Because of the nature of the random bootstrapping, we cannot reproduce serial results exactly in parallel. Our implementation does, however, produce statistically identical results to the serial code.

A typical call to the serial random forest package in R is something like

```
result<-randomForest(x=data, y=classes,
                     ntree=5000,...).
```

The parallelism we are exploiting is in the `ntree` argument, the size of the forest. We distribute the trees evenly amongst all available parallel processes. The subforests are generated in parallel and then combined into a larger forest that is returned on the master process for further analysis. The modification required to an existing analysis script is minimal. We must load the SPRINT library and then just replace calls to `randomForest` by calls to `prandomForest`.

3.1. Combining the results

Subforests generated on each worker process are combined at the end of the run into one large forest that is returned on the master process for further serial analysis. Our first implementation of this used a simple linear algorithm. We gathered all the data onto the master process and combined it there. Benchmarking this algorithm on more than 32 processes demonstrated that a significant amount of time was spent in the combination stage, limiting scalability. Interestingly, sending the data was not the bottleneck but rather the linear complexity algorithm used to combine the results.

The combine operator is associative, and so we would like to take advantage of the reduction algorithms available through MPI. Within MPI libraries, these are typically implemented using a binary-tree-like communication pattern, requiring log *N* steps for *N* processes. Unfortunately, this does not work for our purposes. The object we wish to combine changes size at each level, which means we do not know what value to give as the `count` argument to `MPI_Reduce`.

Because we cannot use built-in MPI reductions, we have implemented a specialised binary-tree reduction within SPRINT that can combine objects of varying size. We use a function signature similar to that of MPI, although the `count` and `datatype` arguments are unnecessary, because R objects are tagged with a type.

```
void reduce(void *in, void *out,
            void *(*combine_fn)(void*, void*),
            int root, MPI_Comm comm).
```

Europe PMC Funders Author Manuscripts

For our use case, the important factor is to minimise the computational cost of combining results, rather than the number of messages. Because of combining the partial random forest objects in `R` code, the time spent in this step is linear in the number of combine operations we carry out. With a gather-to-master approach, this is $O(N)$ as opposed to $O(\log N)$ for the tree-reduction case.

We can imagine extending this technique to all reduction-style operations in a putative future MPI implementation. For programming languages with introspection capabilities, the restriction that processes involved in reduction operations must agree on the `count` and `datatype` arguments [9] is unnecessary. As we have described earlier, in some cases, we would like to carry out a reduction across data of the same type but different sizes.

3.2. Comparison with previous work

The original implementation of random forest is a serial Fortran 77 code [11]. The R implementation available in the `randomForest` package [12] is a faithful transliteration of the original Fortran 77 sources into C with wrappers to the functionality in R. Generation of forests in parallel within R is possible using the `foreach` package [19], and the methods exposed to combine separately generated random forests. More recently, work has been carried out to implement task parallel versions of the algorithm as stand-alone executables. The `parf` project [26] is a task parallel implementation in modern Fortran 90. Other recent work has focused on efficient implementations of the random forest algorithm for the analysis of microarray data—in which the number of variables describing each case is very large, but the number of cases is small. A Java implementation that can deal with very large data sets (in serial) is described in [27]. The `randomjungle` package [28] is another task parallel implementation in C++ expressly designed for microarray data. None of these parallel implementations carries out combination of results in parallel; instead, they all use a gather-to-master approach. As we show later, using a gather-to-master approach for combining subforests introduces a significant bottleneck to parallel performance that is avoided by using a tree reduction.

4. THE RANK PRODUCT METHOD

As previously discussed, in addition to clustering of replicated microarray data, biostatisticians are also interested in feature selection methods to determine or classify ‘important’ genes. For example, is there a statistically significant variation in the gene expression level under two different experimental conditions (e.g. treated or untreated samples)? Early approaches used a fold-change criterion [29], the ratio of the expression level in sample A to that in sample B. However, this method does not allow calculation of significance levels nor is it obvious what the cutoff value for the fold change should be. The rank product method builds on the fold-change criterion but applies it to replicated experiments. That is, rather than just a single sample representing each experimental condition (class), we have many samples in each class.

The rank product method is applicable to experiments comparing two different experimental conditions (class A and class B, say). For each gene, we compute a rank product by ranking the fold-change value of that gene in all pairwise comparison of class A against class B; we

then take the product of these ranks across all samples. The second step is to compute a null distribution for the rank products. This is the expected distribution if there is differentiation between neither genes nor samples. The experimentally observed rank product for each gene can be compared with the null distribution that allows accurate measures of the significance level and estimation of cutoff values [4].

R users can perform rank product analyses with the `RankProd` bioconductor package [13]. These analyses can be performed in parallel using the R packages described in Section 1.1. However, there is no built-in facility to combine separate result objects from the code. If a user wishes to perform the bootstraps in parallel across available processes on their machine, they can but must write code to correctly combine the resulting partial R objects. Our approach in SPRINT parallelises both the bootstrap resampling step and combination of results transparently, returning the combined object to the user for further analysis in serial.

4.1. The algorithm in detail

Consider some microarray data that we may represent by an $N_g \times N_s$ matrix measuring the expression levels of N_g genes in N_s different samples. There are N_a samples from class A and N_b from class B ($N_s = N_a + N_b$). We construct a matrix containing the fold-changes for all pairwise comparisons of class A samples against class B samples (there are $N_a N_b$ of these). Giving us a matrix of fold-changes

$$F = \begin{pmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,N_a N_b} \\ f_{2,1} & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ f_{N_g,1} & \cdots & \cdots & f_{N_g,N_a N_b} \end{pmatrix} \quad (1)$$

where $f_{i,j}$ is the fold-change of gene i in the pairwise comparison j. We note that this step is necessary only for single-channel microarrays. Two-colour arrays measure fold-changes directly (rather than expression levels). These latter data are described as *one-class* as opposed to *two-class* and we analyse them in a similar way. The only difference is that we use the input data as the fold-change matrix directly.

Next, we rank the fold changes in each sample from largest to smallest (for up-regulation) or smallest to largest (for down-regulation). Finally, we compute the rank product of gene i by taking a suitably normalised product of the rank of that gene across all samples

$$r_i = \prod_{k=1}^{N_a N_b} r_{i,k}^{1/(N_a N_b)} \quad (2)$$

where $r_{i,k}$ is the rank of gene i in comparison k.

To obtain significance levels for the experimentally observed rank products (say to determine which genes are statistically strongly up-regulated), we now need to compare this experimentally observed value with the expected distribution of rank products under the null hypothesis. Our null hypothesis is no differential expression of genes— expression levels of individual genes are independent of one another and drawn from the same distribution—and

that all samples are independent (the expression level of gene *i* in sample A is uncorrelated with the level in sample B). Unfortunately, it is not possible to construct an analytic form for the null distribution; we therefore construct it numerically using a bootstrap procedure. We create a random experiment by independently permuting each sample's gene expression vector and calculate the rank product of all the genes in this random data. We repeat this many times to build a distribution of rank products for the null hypothesis.

4.1.1. The null distribution in the large-sample limit—The rank product (r_i) under the null distribution looks suspiciously like the product of independent random variables, and we may therefore be tempted to apply the central limit theorem to its logarithm to construct an approximate analytic distribution. We note that this approach *does* work for data obtained from two-colour arrays that measure fold changes directly. For a large number of samples, we expect the logarithm of r_i to be distributed normally with mean $\mu = 1/N_g \Sigma_{i=1}^{N_g} log\, i$ and variance $\sigma^2 = 1/N_s \left(1/N_g \Sigma_{i=1}^{N_g} log^2 i - \mu^2\right)$. We note that this approximation is only really useful in the $N_s >> 1$ limit and as such our implementation just bootstraps the two-colour array distribution in the same way as the single-channel data, because this approach does not work for single-channel arrays anyway. Although the gene expression levels in the null hypothesis are independent, computation of the pairwise fold changes introduces correlations which mean that the $r_{i,k}$ are not independent. See also [30] for a different approach to obtaining the null distribution for two-colour microarray data.

4.2. Parallelisation approach

The computationally expensive part of rank product analysis is the generation of the bootstrapped null distribution. This is not a memory bound problem because we are only ever interested in summary statistics of the form $\int_{-\infty}^{r_i} P(x)\, dx$, where r_i is the observed rank product of gene *i* and $P(x)$ is the bootstrapped null distribution. In parallel, rather than first computing the full bootstrapped $P(x)$ and subsequently evaluating the integral, we can instead keep a running count for each gene. The memory requirements of this latter approach are $O(N_g)$ rather than $O(N_g B)$ (B the number of bootstrap samples). This deviates from the existing serial implementation in `RankProd` package, which first computes the full boostrapped distribution $P(x)$ and then the summary statistics. Our approach is more memory efficient and therefore performs approximately twice as fast as the `RankProd` package in serial.

Our implementation takes a task parallel approach and divides up the requested number of bootstrap samples between available processes. The input data set is broadcast to all processes, bootstraps are calculated independently and the results are then collated and returned to the master process for further analysis. As long as the input data set fits in the available memory, this approach works well. For very large data sets, a data parallel approach is probably necessary. We describe one possible approach in Section 4.3.

4.3. Dealing with large data

Our parallel implementation of the rank product method assumes that the data can be analysed serially (it is just time-consuming). Although this is true for current generation

microarray data, future sequence data may not meet this restriction. Where microarray platforms reach millions of measurements and thousands of samples, the data may no longer fit in RAM. We would then have to parallelise across a single data set, just to perform the analysis. In this section, we describe a possible implementation strategy for this case.

Recall that our data is a matrix with N_g rows (each one corresponding to a single expression measurement) and N_s columns (each column being one sample). We can choose to distribute the data either by assigning contiguous rows or contiguous columns to parallel processes. Dividing the data by row is the better option, typically $N_g >> N_s$ and we can therefore divide the problem between more processes if we parallelise across N_g. Furthermore, calculation of fold changes requires that we have access to all data in a row. If we parallelise across columns, we must communicate all the row data for every bootstrap sample.

Having divided up the rows between processes, we can compute the fold changes for our row locally. Ranking the genes requires that we sort the columns. This must be carried out in parallel. Parallel sorting is a well-studied research topic (see [31] for an early overview, and [32, 33] for more modern methods). Once the columns have been sorted, the rank product for each row can be obtained without further communication. Construction of the bootstrap sample also requires communication: we must permute columns in parallel. This can be implemented using a parallel sort on a random key. The parallel implementation would therefore be

- Distribute the input data row-wise.
- Sort each column in parallel to rank the actual data.
- Locally compute the rank product for each row.
- Construct a bootstrap sample by permuting each column in parallel.
- Sort each column to rank the bootstrap data.
- Compute the bootstrapped rank product.
- Repeat bootstrap step as necessary.

This method should work irrespective of the data size, although it will doubtless be slower than a task parallel approach.

5. PERFORMANCE OF THE PARALLEL IMPLEMENTATIONS

5.1. Benchmark system

The parallel functionality was tested and benchmarked on the UK national supercomputing service, HECToR. Both the Cray XT4 (quad core AMD Opteron nodes) and Cray XT6 (24 core AMD Magny-Cours nodes) incarnations were used. We found essentially no variation in performance between the two systems; as a result, graphs only show results from the XT4 system.

5.2. Rank products

Parallelisation of the bootstrap starts with a broadcast of the input data to slave processes. Calculation of the partial distribution of rank products requires no communication. Finally, further communication is required to gather the partial counts to the master process. We therefore expect that the scaling performance should be good: we should only see a drop-off in scaling when the number of bootstrap samples per process becomes very low. The wall clock timings we report here include a serial section: calculation of the empirical rank product values from the input data. The maximum possible parallel efficiency on p processes relative to a serial calculation is therefore

$$\eta_p = \frac{b+1}{b+p} \quad (3)$$

where b is the required number of bootstrap samples. We compare the expected parallel efficiency with that obtained by our parallelisation scheme in Figure 1. Execution time on a single core is 26 000 seconds for 1024 bootstrap samples and estimated at 434 000 seconds for 16 384 samples. In comparison, the original `RankProd` code requires 42 300 seconds for 1024 samples.

5.3. Random forests

In our implementation of a parallelised random forest algorithm, we used a data set from a biological study within the Division of Pathway Medicine, University of Edinburgh. In order to test the hypothesis that blood-borne infection in young infants can be identified via gene transcription profiling, this study consists of blood samples from 62 infants, 27 of which have a confirmed bacteriological infection and 35 are non-infected controls [34]. The blood samples were processed to RNA level and each sample hybridised to an Illumina[¶] human gene expression microarray. Each array contains approximately 23 000 probe sequences that measure the expression of all known genes in the human genome. With classification algorithms, the overall goal is to identify sets of genes that can reliably identify an unknown blood sample as infected or not infected. Larger data sets were synthesised by replicating (with noise) either the rows or columns (where appropriate) of our input data.

We first present the effects that implementing a parallel combining step has on the runtime of our algorithm. Figure 2 shows the fraction of the total runtime spent combining results for both the serial gather-to-master approach and the parallel tree-reduction. We see that the tree approach gives significantly better performance. This is not because of communications latency, all the data for the final result can be gathered to the master process in a fraction of the time spent combining them.

Figure 3 shows the speedup of our parallel implementation compared with an optimal (linear speedup). We see good speedup to 128 cores for this size of problem before the overhead of communicating the partial results and combining them outweighs any performance gains. As shown in Figure 2 with an $O(\log N)$ algorithm for combing results, the percentage of the

[¶] http://www.illumina.com

total runtime spent creating the final complete random forest is almost 50% when utilising 256 processes.

Finally, Figure 4 shows the performance of our code when increasing both the size of the data and the number of processes (weak scaling). We note in this last instance the poor performance of the code when the number of genes in the data set becomes very large. This is a previously observed issue [35,36] with the serial implementation we have used. Although the parallel efficiency for large numbers of genes is poor, it does significantly reduce the time to solution for large data sets. As an example, producing a random forest with 8192 trees for data with 51 2000 measurements and 62 cases takes almost 2 hours in serial. With our parallel implementation, we can generate the same forest in 63 seconds with 512 parallel processes. This allows significantly faster exploratory data analysis, because the biostatistician no longer needs to wait 2 hours for an analysis to complete. This becomes especially useful when frequent re-runs are necessary for parameter optimisation or problem solving.

6. CONCLUSIONS AND FUTURE WORK

We have implemented a task parallel version of the random forest algorithm for use in R using the SPRINT library. For typical use cases, we can obtain a speedup of around 40 over the same serial code on HECToR using 128 processes. Rather than implementing the algorithm from scratch, we used an existing serial implementation and added a parallel wrapper around it. Unlike other task parallel packages for R, we also implemented a tree-reduction algorithm to combine results in parallel rather than serial. This had a surprisingly large effect on the overall performance. Our interface exactly mimics the existing serial implementation: modifying existing serial R scripts to take advantage of this functionality is trivial. Because combining trees does not significantly increase memory use, this parallelisation technique may be sufficient to alleviate the problems using random forest classification for genome wide association studies reported in [37].

Acknowledging that classification problems on large data sets may be preceded by feature selection problems within the same data, we have implemented a parallel version of the rank product feature selection method. For a small number of bootstrap samples, we obtain speedups of around 400 (on 256 processes) over the publicly available `RankProd` package —approximately half of this factor coming from the more efficient serial implementation, we have developed, the rest from parallelisation of the boostrap resampling. For large bootstrap counts, the obtainable speedup is even greater (around a factor of 8000 on 8192 processes). We also propose a data parallel implementation of the rank product method should microarray data become too large to conveniently process even a single bootstrap sample serially.

Our work shows that adding task parallel functions to SPRINT is quite straightforward and can provide significant performance gains over serial codes. We do, however, have to pay attention to all parts of the implementation including distributing the data and gathering the results: it is often not enough to just speed up the main loop of the code.

6.1. Future work

The serial random forest implementation available in R was written with data from the social sciences in mind: large numbers of cases and few variables per case. This is the exact opposite of the type of data seen in microarray analysis. As a result, the code is not very efficient when dealing with microarray data, especially when the number of variables becomes very large.

These issues with large microarray data have previously been reported in the literature [27, 35-37]. The random jungle package [28] has been expressly written to avoid these problems. At present, no R interface exists to this new package. If an R interface to this package were developed, it could be simply incorporated into SPRINT: we would only require changes to a few interface functions.

Similarly, our rank product implementation has been written with an eye to current and nextgen sequence data, for which we expect that the task parallel approach will be sufficient. Should future generation sequence data reach multi-GB sizes (millions of measurements and thousands of samples), our implementation may struggle. We propose a data parallel implementation of the method in Section 4.3 that should scale irrespective of the data size.

Acknowledgments

The SPRINT project is funded by a Wellcome Trust grant [086696/Z/08/Z]. Data were kindly provided by Thorsten Forster of the Division of Pathway Medicine, University of Edinburgh.

This project was funded under the HECToR distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR—A Research Councils UK High End Computing Service—is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE support Service is provided by NAG Ltd. http://www.hector.ac.uk

REFERENCES

1. The chipping forecast II. Nature Genetics. 2002; 32(4s):461–552.
2. Clarke R, Ressom HW, Wang A, Xuan J, Liu MC, Gehan EA, Wang Y. The properties of high-dimensional data spaces: implications for exploring gene and protein expression data. Nature Reviews Cancer. 2008; 8(1):37–49.
3. Wang Y, Miller DJ, Clarke R. Approaches to working in high-dimensional data spaces: gene expression microarrays. British Journal of Cancer. 2008; 98(6):1023–1028. [PubMed: 18283324]
4. Breitling R, Armengaud P, Amtmann A, Herzyk P. Rank products: a simple, yet powerful, new method to detect differentially regulated genes in replicated microarray experiments. FEBS Letters. 2004; 573:83–92. DOI: 10.1016/j.febslet.2004.07.055. [PubMed: 15327980]
5. Dudley J, Pouliot Y, Chen R, Morgan A, Butte A. Translational bioinformatics in the cloud: an affordable alternative. Genome Medicine. 2010; 2(8):51. [PubMed: 20691073]
6. Giancarlo R, Scaturro D, Utro F. Computational cluster validation for microarray data analysis: experimental assessment of clest, consensus clustering, figure of merit, gap statistics and model explorer. BMC Bioinformatics. 2008; 9(1):462. [PubMed: 18959783]
7. Hill J, Hambley M, Forster T, Mewissen M, Sloan T, Scharinger F, Trew A, Ghazal P. SPRINT: a new parallel framework for R. BMC Bioinformatics. 2008; 9(1):558. [PubMed: 19114001]
8. Mewissen, M. SPRINT – user requirements survey results. Division of Pathway Medicine, University of Edinburgh; 2010. Technical Report
9. Message Passing Interface Forum. MPI: a message-passing interface standard version 2.2. 2009.

10. Petrou S, Sloan TM, Mewissen M, Forster T, Piotrowski M, Dobrzelecki B, Ghazal P, Trew A, Hill J. Optimization of a parallel permutation testing function for the SPRINT R package. Concurrency and Computation: Practice and Experience. 2011; 23(17):2258–2268. DOI: 10.1002/cpe.1787. [PubMed: 23335858]
11. Breiman L. Random forests. Machine Learning. 2001; 45:5–32.
12. Liaw A, Wiener M. Classification and regression by randomForest. R News. 2002; 2(3):18–22.
13. Hong F, Breitling R, McEntee CW, Wittner BS, Nemhauser JL, Chory J. RankProd: a bioconductor package for detecting differentially expressed genes in meta-analysis. Bioinformatics. 2006; 22(22):2825–2827. DOI: 10.1093/bioinformatics/btl476. [PubMed: 16982708]
14. Ihaka R, Gentleman R. R: a language for data analysis and graphics. Journal of Computational and Graphical Statistics. 2005; 5(3):299–314.
15. R Development Core Team. R: a language and environment for statistical computing, R Foundation for Statistical Computing. Vienna, Austria: 2010. Available from: http://www.R-project.orgISBN 3-900051-07-0 [accessed 1st September 2011]
16. Yu, H. [accessed 1st September 2011] Interface (wrapper) to MPI (Message-Passing Interface). r package version 0.5-9. 2010. Available from: http://cran.r-project.org/package=Rmpi
17. Vera G, Jansen R, Suppi R. R/parallel – speeding up bioinformatics analysis with R. BMC Bioinformatics. 2008; 9(1):390. [PubMed: 18808714]
18. Samatova, NF.; Bauer, D.; Yoginath, S.; Task-parallel, R. [accessed 1st September 2011] r package version 0.34. 2009. Available from: http://cran.r-project.org/package=taskPR
19. REvolution Computing. [accessed 1st September 2011] foreach: Foreach looping construct for R. r package version 1.3.0. 2009. Available from: http://CRAN.R-project.org/package=foreach
20. Breiman, L. Classification and Regression Trees. Wadsworth; Belmont, California: 1984.
21. Quinlan JR. Induction of decision trees. Machine Learning. 1986; 1(1):81–106.
22. Kotsiantis SB. Supervised machine learning: a review of classification techniques. Informatica. 2007; 31:249–268.
23. Joshi, MV.; Karypis, G.; Kumar, V. ScalParC: a new scalable and efficient parallel classification algorithm for mining large datasets; Proceedings of the International Parallel Processing Symposium; Orlando, Florida, USA. 1998; p. 573-579.
24. Joshi, MV.; Karypis, G.; Kumar, V. Technical Report. University of Minnesota; 1998. ScalParC: a new scalable and efficient parallel classification algorithm for mining large datasets. Available from: http://glaros.dtc.umn.edu/gkhome/fetch/papers/scalparc.pdf [accessed 1st September 2011]
25. Shafer, JC.; Agrawal, R.; Mehta, M. SPRINT: a scalable parallel classifier for data mining. In: Vijayaraman, TM.; Buchmann, AP.; Mohan, C.; Sarda, NL., editors. VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996. Morgan Kaufmann; Mumbai (Bombay), India: 1996. p. 544-555.
26. Topiẃc, G.; Smuc, T.; Sojat, Z.; Skala, K. Reimplementation of the random forest algorithm. In: Vajtersic, M.; Trobec, R.; Zinterhof, P., editors. Parallel Numerics '05. University of Salzburg; Salzburg, Austria: 2005. p. 119-125.
27. Manilich, EA.; Ozsoyoglu, ZM.; Trubachev, V.; Radiovoyevitch, T. Classification of large microarray data sets using fast random forest generation; Proceedings of the 9th Computational Systems Bioinformatics Conference; Stanford, California, USA. 2010; p. 82-91.
28. Schwarz DF, König IR, Ziegler A. On safari to Random Jungle: a fast implementation of Random Forests for high-dimensional data. Bioinformatics. 2010; 26(14):1752–1758. [PubMed: 20505004]
29. DeRisi JL, Iyer VR, Brown PO. Exploring the metabolic and genetic control of gene expression on a genomic scale. Science. 1997; 278(5338):680–686. DOI: 10.1126/science.278.5338.680. [PubMed: 9381177]
30. Koziol JA. Comments on the rank product method for analyzing replicated experiments. FEBS Letters. 2010; 584(5):941–944. [PubMed: 20093118]
31. Akl, SG. Parallel Sorting Algorithms. Academic Press; Orlando, Florida: 1985.
32. Shi H, Schaeffer J. Parallel sorting by regular sampling. Journal of Parallel and Distributed Computing. 1992; 14:361–372.

33. Helman DR, Jájá J, Bader DA. A new deterministic parallel sorting algorithm with an experimental evaluation. Journal of Experimental Algorithsm. 1998; 3:4.

34. Smith CL, Dickinson P, Craigon M, Ross A, Khondoker MR, Forster T, Ivens A, Lynn DJ, Orme J, Jackson A, Lacaze P, Stenson BJ, Ghazal P. Host immune-metabolic network response detecting human neonatal bacterial infection. Journal of Clinical Investigation. 2011 (submitted).

35. Ziegler A, DeStefano AL, König IR. Data mining, neural nets, trees – problems 2 and 3 of genetic analysis workshop 15. Genetic Epidemiology. 2007; 31(Supplement 1):S51–S60. [PubMed: 18046765]

36. Schwarz D, Szymczak S, Ziegler A, Konig I. Picking single-nucleotide polymorphisms in forests. BMC Proceedings. 2007; 1(Suppl 1):S59. [PubMed: 18466559]

37. Meng Y, Yu Y, Cupples LA, Farrer L, Lunetta K. Performance of random forest when SNPs are in linkage disequilibrium. BMC Bioinformatics. 2009; 10(1):78. DOI: 10.1186/1471-2105-10-78. [PubMed: 19265542]

Europe PMC Funders Author Manuscripts

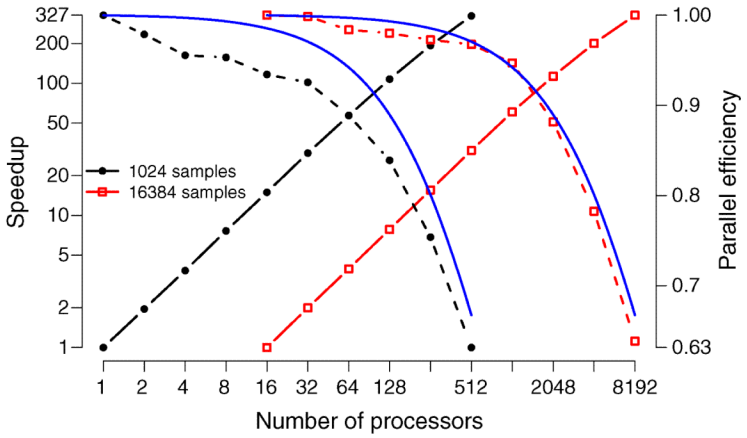

Figure 1. Speedup (solid lines) and parallel efficiency (dashed lines) for two-class rank product analysis, blue (solid) lines show theoretical peak efficiency (Equation 3).

Data set has 23 292 genes and 35 + 27 samples, leading to 945 pairwise comparisons.

Europe PMC Funders Author Manuscripts

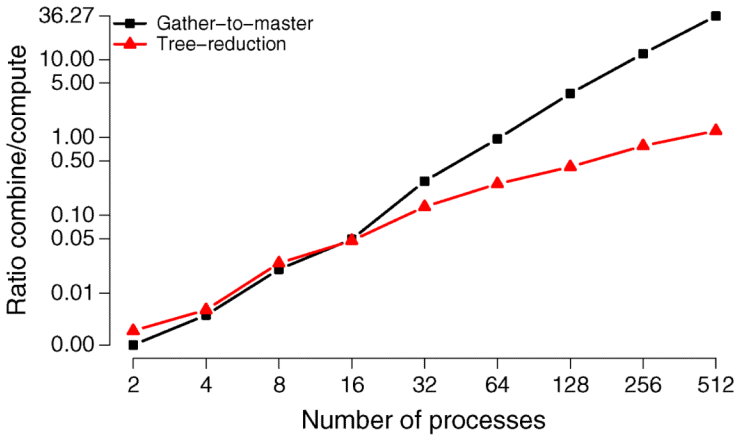

Figure 2. Ratio of time spent combining results to time computing trees as the number of parallel processes is increased.

For large process counts the linear gather-to-master approach (black squares) performs much worse than a tree reduction (red triangles). The gather to master approach is (N) in the number of processes, the tree reduction is $O(\log N)$. Timings are for generation of 8192 O trees with 23 292 genes and 62 cases.

Europe PMC Funders Author Manuscripts

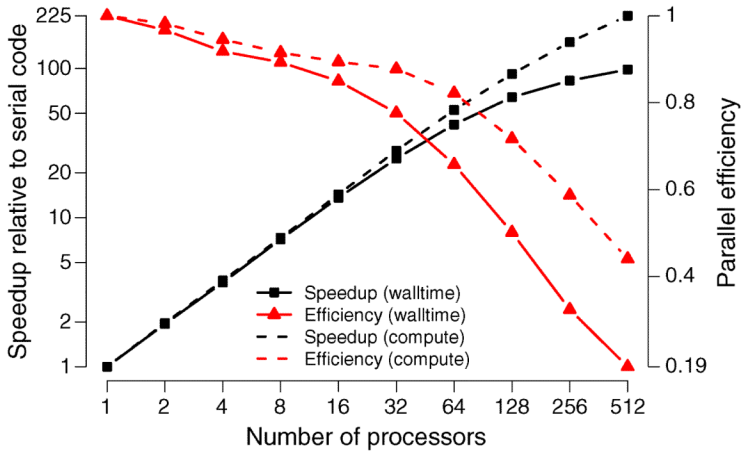

Figure 3. Speedup (black squares) and parallel efficiency (red triangles) for the generation of a forest with 8192 trees for a data set with 23 292 genes and 62 cases.

Dashed lines show speedup and parallel efficiency if we exclude the cost of gathering and combining the results back to the master process (i.e. compute only). Overall parallel efficiency is good up to 128 processes after which the costs of communicating and combining the partial results outweigh gains in computation.

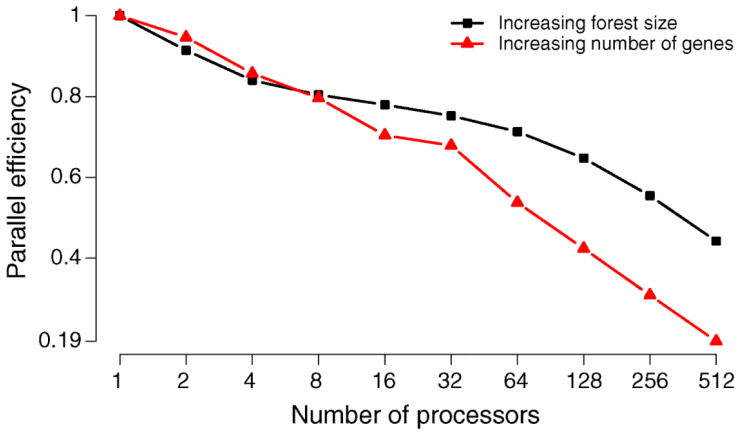

Figure 4. Parallel efficiency with increasing total problem size (fixed problem size per processor).

Black squares show the efficiency with increasing forest size (23 292 genes and 62 cases, 128 trees per process). Red triangles show the efficiency with increasing numbers of genes for a fixed forest size (8192 trees, 1000 genes and 62 cases per process), note significant drop-off in efficiency when the data set gets very large.

Europe PMC Funders Author Manuscripts